McGill University
School of Computer Science
Sable Research Group

A portable and high-performance matrix operations library for CPUs, GPUs and beyond

Sable Technical Report No. sable-2013-1

Rahul Garg and Laurie Hendren

30 April 2013

Contents

1 Introduction **3**

2 OpenCL background **4**

3 General design of RaijinCL **5**

3.1 Autotuning 5

3.2 Easy deployment 5

3.3 Asynchronous API 5

3.4 Control over resource allocation 5

4 Autotuning process and kernel design **6**

4.1 Kernels for general matrix-matrix multiply (GEMM) 6

4.1.1 GEMM Background 6

4.1.2 Codelets and search parameters 7

4.2 Kernels for general matrix-vector multiply (GEMV) 8

4.3 Kernels for transpose 9

5 Experimental results and analysis **9**

5.1 Architecture Tested 14

5.2 Discussion of Results 14

6 Related Work **16**

7 Conclusions **17**

List of Figures

1 Tiles computed by a work-item, consecutive or offset elements 8
2 GPU performance for SGEMM 11
3 GPU performance for DGEMM 12
4 GPU performance for CGEMM 13
5 SGEMM performance on Evergreen on NT kernel, internally converted to TN by RaijinCL . 15

List of Tables

1 Optimal Parameters for SGEMM, DGEMM and CGEMM 10
2 Percentage of peak obtained on each device in best case 12

Abstract

High-performance computing systems today include a variety of compute devices such as multi-core CPUs, GPUs and many-core accelerators. OpenCL allows programming different types of compute devices using a single API and kernel language. However, there is no standard matrix operations library in OpenCL for operations such as matrix multiplication that works well on a variety of hardware from multiple vendors. We implemented an OpenCL auto-tuning library for real and complex variants of general matrix multiply (GEMM) and present detailed performance results and analysis on a variety of GPU and CPU architectures. The library provides good performance on all the architectures tested, and is competitive with vendor libraries on some architectures (such as AMD Radeons) We also present brief analysis for related kernels such as matrix transpose and matrix-vector multiply.

1 Introduction

Hardware trends in fields ranging from supercomputing to mobile computing point to a future where different types of compute devices (CPUs, GPGPUs, DSPs and many-core processors like Xeon Phi) co-exist. We refer to non-CPU devices as accelerators for brevity. Dense matrix operations are an important use-case for accelerators and operations like general matrix-multiply (GEMM) are particularly important. Application programmers would prefer their code to be portable across machines. Portability of code requires a portable programming language as well as uniform APIs for important libraries. On CPUs, standards have been formed for programming languages (such as Fortran and C/C++). For dense matrix operations on CPUs, users can rely on the BLAS (basic linear algebra subsystem) [7] application programming interface (API). BLAS implementations are available from many vendors and open-source solutions such as ATLAS [3] and OpenBLAS are also available.

The problem of portability across compute devices is still being solved. OpenCL [1] is emerging as a common, portable programming language across various compute devices and implementations are already available from many vendors for a variety of compute devices (CPUs, GPGPUs, Xeon Phi, FPGAs and DSPs). However, there is no standardized OpenCL BLAS API, nor any high-performance open-source portable solution. For example, AMD provides an OpenCL BLAS [2] but it does not perform well on non-AMD hardware, while Nvidia's CUBLAS library [4] is only available to users of Nvidia's proprietary CUDA toolkit. In absence of a standardized OpenCL BLAS API, users may think about writing their own OpenCL kernels for operations like GEMM. OpenCL's programming model enables two features necessary for obtaining high performance: expressing parallelism, and a portable SIMD abstraction through built-in short-vector types such as float4 types. However, these features are not sufficient for achieving high performance. Important details like memory hierarchy and thread dispatch mechanics vary widely between architectures and hence a single OpenCL kernel does not offer great performance across architectures.

We provide a solution to the challenge of a vendor-neutral, high-performance OpenCL library for matrix operations. We have designed and implemented an OpenCL library called RaijinCL that provides high-performance implementations of general matrix-matrix multiply (GEMM), general matrix-vector multiply (GEMV) and matrix transpose. The library has been tested on GPUs and CPUs from multiple vendors such as AMD, ARM, Intel and Nvidia. The library is an auto-tuning library that tests various types of kernels and tuning parameters on each compute device in a user's machine, and caches the best found kernel for each compute device. The library is available as open-source and we have received many queries about the library from both academic and commercial users, indicating that the library is of high interest to the community.

We make four contributions in this paper. First, we describe the design and implementation of our auto-tuning library. We describe the basic design decisions of the library, as well as the set of kernels and parameter space explored by the library for GEMM, GEMV and transpose kernels. Second, we present performance results from a range of architectures (both CPUs and GPUs, from low-power to high end) from multiple vendors on SGEMM, DGEMM and CGEMM. We show that GEMM routines selected by our library achieve good performance in every case, and are competitive with the vendor's proprietary BLAS on various GPUs. We are the first to present results on Intel's integrated GPUs on Ivy Bridge processors. Our third contribution is the analysis of performance of the explored kernels on the tested architectures. We base our analysis upon the architectural disclosures made by vendors and the disassembly of the OpenCL kernels where available. Our fourth contribution is the discussion of good API design for an OpenCL auto-tuning BLAS library. We describe design decisions such as ease of installation, managing resource-usage and asynchronous design.

The structure of the paper is as follows. Some key background about OpenCL is given in Section 2, and the design of our library is presented in Section 3. The details of GEMM kernels are presented in Section 4.1, GEMV kernels in Section 4.2 and transpose kernels in Section 4.3. Performance results and analysis are presented in Section 5. Related literature is described in Section 6 and we present our conclusions in Section 7.

2 OpenCL background

OpenCL offers a SPMD-like parallel programming model. Programmers write *kernel functions* in a C99-derived kernel language and then dispatch it across a number of work-items. A work-item can be thought of as a lightweight or fine-grained thread. Each work-item executes the same kernel function, but each work-item has its own independent control flow. Work-items are organized into work-groups. Items within a work-group can read/write from a small pool of memory (typically a few kilobytes) that is called *local memory*. Work-items within a work-group can also synchronize with each other through the user of barriers. However, work-items from different work-groups cannot synchronize with each other.

OpenCL kernel functions operate in their own address space. OpenCL requires the programmers to allocate OpenCL memory objects and OpenCL provides API functions to transfer data between the OpenCL memory objects and the regular application data structures. On discrete GPUs and accelerators such as Xeon Phi, OpenCL memory objects will typically reside on GPU's on-board high-speed memory. OpenCL offers two types of memory objects: buffers and images. Buffers are similar to linear arrays in C. Image data-types, as the name implies, are intended for image data and are less flexible than the buffer data-type. On GPUs, memory operations on images are performed through specialized texture-sampling units. The texture samplers are generally optimized for certain read/write patterns and may have their own caches. It is sometimes beneficial to store non-image data as image data-types because it gives us access to the texture-sampling hardware.

So far we have described OpenCL functionality relevant to one OpenCL-capable device in a machine. A machine may have more than one OpenCL capable device and each device is exposed separately to the programmer. Consider a machine with a CPU and a GPU. Let us assume we want to execute an expensive computation on the GPU. While the GPU is busy computing, for best application performance the CPU should not sit idle waiting for the result. OpenCL offers an asynchronous design to solve this problem. OpenCL kernel calls do not block the execution of the program. Instead, kernels are enqueued into a device-specific work-queue. The API call to enqueue kernel calls returns an event object. The program can enqueue a kernel call and then proceed with other useful work. Later, the application can query the status of the event, choose to wait until the event is complete or register a callback function to be executed whenever the event is complete. RaijinCL builds upon OpenCL's asynchronous API design style.

3 General design of RaijinCL

RaijinCL has twin goals of being portable, and being usable in the real-world. Accordingly, we have adopted the following design decisions.

3.1 Autotuning

OpenCL is a portable API but it is not performance-portable. The same OpenCL kernel does not work well across devices. Thus, for each problem, RaijinCL implements a number of parameterized OpenCL kernel code generators called *colts*. Each modeled represents a particular algorithmic variation for the problem. For example, we have six codelets for GEMM that are described in Section 4.1. Each codelet may be parameterized with a number of integer and boolean parameters such as the vector length and work-group size. If there are N parameters, we get a N-dimensional search space associated with each codelet. A N-dimensional point in the search space can be passed to the codelet's code generator and results in a unique OpenCL kernel. RaijinCL generates a kernel for all valid combinations of codelets and points in the search space. The performance of all generated kernels is measured and RaijinCL stores the best performing kernel.

3.2 Easy deployment

One issue with autotuning libraries is that deployment of the library can be hard. We have tried to simplify this task. RaijinCL itself can be distributed in binary form and comes with a small command-line utility for performing autotuning. The user specifies the device to tune for from the list of OpenCL capable devices installed on his/her machine. The tuning application only requires the OpenCL driver and does not require the user to install a C/C++ compiler. The tuning application generates and tests many different OpenCL kernels, and creates a device profile for the specified device. The device profile contains the best found OpenCL kernels as well as some metadata. The device profile is a simple self-contained text file and can be easily redistributed for deployment by other users of the same device. Device profiles for some popular devices like AMD Tahiti (which powers GPUs such as Radeon 7970) are available on our website. If a device profile is already available for your device, then you can simply download the profile and skip the tuning process completely. We are hoping that the community and hardware vendors will contribute many device profiles which will further simplify the deployment process.

3.3 Asynchronous API

Following OpenCL's design principles outlined in Section 2, RaijinCL's API is also asynchronous. Computationally heavy API calls in RaijinCL, such as GEMM, perform minimal setup and simply enqueue relevant kernel calls to the OpenCL device. Thus, RaijinCL API calls finish very fast without blocking the CPU and return an OpenCL event object.

3.4 Control over resource allocation

Efficient implementation of GEMM routines can require copying the arguments into a more efficient layout into a temporary buffer. Usually the GEMM library implementation will allocate, use and destroy the temporary buffers automatically without exposing them to the application programmer. RaijinCL offers both a high-level API, that is similar to other GPU BLAS APIs such as AMD's OpenCL BLAS, and a low-level API. The high-level API implementation automatically allocates the temporary buffers, performs the appropriate copy or transpose operation and registers a callback with OpenCL to destroy the buffers when the kernel call is complete.

However, there are two performance related concerns with this high-level approach. First, consider a sequence of three matrix multiplication calls $A \cdot B$, $A \cdot C$, and $A \cdot D$. Here, the GEMM library will perform the allocate-copy-destroy operation for A three times which is inefficient. Second, memory available to discrete GPU devices is often very limited and thus the application may want to know the exact memory usage of library routines and may want to explicitly manage the life-cycle of the memory objects allocated by the library. This is difficult to achieve in a high-level API. In the case of our high-level API, OpenCL runtime does not provide a guarantee of the amount of delay between the finishing of the kernel call and execution of the callback to destroy the buffers. Thus, in addition to the high-level API, we offer a four-step low-level API. First routine in the low-level API determines the size and type of temporary buffers required for a given input size and layout, and allocates them. The buffers may be reused for multiple problems of the same size and layout. Second, the programmer calls the copy routine. Third, the computation kernels are called. Finally, the temporary buffers can be deallocated.

Thus, RaijinCL offers a choice between the convenience of a high-level API and control of a low-level API.

4 Autotuning process and kernel design

To implement the RaijinCL autotuner we have identified the key parameters that affect performance of each kernel and designed a set of parameterized codelets that cover the design space. For example, RaijinCL implements six codelets for SGEMM and DGEMM, and six different codelets for CGEMM. The codelets are parameterized over a variety of different features, for example the work-group size. An important part of the design of RaijinCL was determining the correct codelets and parameters, so that the right design space is exposed for each kernel.

RaijinCL's autotuner generates kernels for each valid combination of codelet and search parameters and measures the performance on a particular problem size (2048x2048 by default). The best performing kernel and metadata (such as work-group size) required for execution of the kernel is stored in the device profile. Once the tuning process is over, calls to RaijinCL APIs will utilize the kernels stored in the device profile. If the input matrices happen to be much larger than the tuning size, then the input matrices are tiled into smaller matrices of the tuned size.

4.1 Kernels for general matrix-matrix multiply (GEMM)

In this section we first give some background on GEMM and discuss general ideas, and then we discuss the codelets and search space for RaijinCL's GEMM algorithms.

4.1.1 GEMM Background

General matrix-multiply (GEMM) computes the operations $C = \alpha op(A)op(B) + \beta C$ where A, B and C are matrices, α and β are scalars. $op(A)$ is either A or A^T, and $op(B)$ is either B or B^T depending on input flags specified by the caller. Our GEMM API supports both row-major and column-major orientations. However, for this paper, we only consider row-major layouts. We support three possible datatypes for the elements of A, B and C: single-precision floating point, double-precision floating point and complex double-precision floating point which correspond to SGEMM, DGEMM and CGEMM in the BLAS terminology.

Four variations of GEMM can be considered: $C = \alpha AB + \beta C$, $C = \alpha A^T B + \beta C$, $C = \alpha AB^T + \beta C$ and $C = \alpha A^T B^T + \beta C$. These are called the NN, TN, NT and TT kernels respectively where N corresponds to no transpose and T corresponds to transpose. For square matrices the memory read pattern in TT kernel is very similar to the NN kernel, and we focus on NT, TN and NN layouts only for this paper. Let us assume we have an efficient kernel for any one of the three cases, and we have efficient transpose and copy routine. Then, one need not find efficient routines for the remaining layouts. One can simply transpose or copy the inputs appropriately, and then call the most efficient kernel. However, the memory access pattern for the TN, NT and NN cases can be quite different from each other and the layout found to perform the best on one architecture may not be the best layout on another architecture. Thus, we have implemented three variations of matrix multiplication: TN, NT and NN.

A naive row-major NN matrix-multiply kernel is shown in Listing 1. A naive OpenCL implementation will assign computation of one element of C to one work-item. However, such an implementation will make poor use of the memory hierarchy of current compute devices. Thus, typically matrix multiplication is tiled. Each of the three loop directions i,j and k, can be tiled with tiling parameters T_i, T_j and T_k and each work-item is assigned to compute a tile $T_i \times T_j$ of C. This tile is computed as a sum of a series of matrix multiplications of $T_i \times T_k$ and $T_k \times T_j$ tiles of A and B respectively.

```
int i,j,k;
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
      for(k=0;k<K;k++){
          C[i][j] += A[i][k]*B[k][j];
      }
  }
}
```

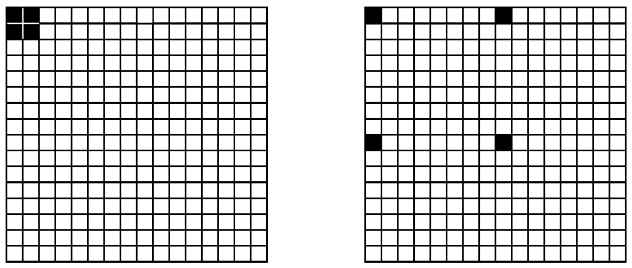
Listing 1: Row-major NN matrix-multiply

Consider a work-group of size W_x, W_y, where each work-item computes a tile of size (T_i, T_j). The work-group will compute a $(W_x \times T_i, W_y \times T_j)$ tile of C. While we have specified the tile size, we have not specified how the tile-elements are assigned to work-items. We have implemented two possible assignments. The first assignment is that each work-item computes a tile consisting of T_i consecutive rows and T_j consecutive columns. The second possibility is that the T_i rows are offset by W_x from each other, and T_j tiles are offset by W_y from each other. We give a visual example. Consider a work-group of size $(8, 8)$ where each work-group is assigned $(2, 2)$ tile. Then, two possibilities for elements computed by the $(0, 0)$ work-item are shown in Figure 1.

4.1.2 Codelets and search parameters

Our autotuner has six codelets for GEMM. Each codelet implements a particular layout (NT, TN or NN) and a particular element assignment scheme (consecutive or offset), thus giving six combinations. Each codelet has the following parameters:

Tile sizes: A tuple of three values $(T_i, T_j, T_k$, indicating the tile size in the three loop directions.



(a) consecutive elements (b) offset elements

Figure 1: Tiles computed by a work-item, consecutive or offset elements

SIMD width: All loads and multiply-accumulate operations are done according to the value of this parameter. Our code generator explores SIMD widths of 1,2,4 and 8 for SGEMM and 1,2 and 4 for DGEMM.

Work-group size: The number of work-items inside the group. We currently search for four possibilities: $(8,8)$,$(16,4)$,$(4,16)$ and $(16,16)$. We chose this set because most current GPU optimization manuals recommend work-group size be 64 or a multiple of 64.

Local memory usage: Each input matrix can be brought into a workgroup's local memory, which is shared across work-items, or fetched directly from global memory. Thus, there are two possible options for each input, and four possible values for this parameter.

Use of OpenCL images: Storing the input matrices as OpenCL images may be beneficial on some GPU architectures. Again, each input may or not be stored as an image, and thus there are four possible values for this parameter.

On CPUs, we restrict the search space based upon common CPU performance characteristics. First, we restrict our search to codelets with NT layout. Other layouts access the matrices column-wise, accessing a large number of pages which may lead to translation look-aside buffer (TLB) misses. Second, on CPUs our autotuner skips kernels that use OpenCL images because CPUs do not have dedicated texturing hardware and thus storing data as images will not provide any benefit. Finally, our autotuner skips kernels using local memory for loading inputs A or B on CPU devices because CPU devices lack dedicated local memory hardware.

4.2 Kernels for general matrix-vector multiply (GEMV)

The general matrix-vector multiply (GEMV) operation is defined as follows: $y = \alpha \cdot op(A) \cdot y + \beta \cdot y$, where $op(A)$ is either A or A^T, x and y are vectors and α and β are scalars. Vectors x and y need not be contiguous, they can be strided with specified strides. We have a generic untuned implementation of GEMV that handles all the cases. We have also generated an autotuned optimized codepath for the case where x and y are contiguous.

Our autotuner considers several codelets. In the first codelet, the autotuner considers implementation where one work-item computes one output element. For example, consider a matrix of size 2048×2048 being multiplied by a vector of size 2048. In this case, we can launch a total of 2048 work-items. However, such a strategy may not have enough parallelism to be a good fit for GPUs and may not exploit memory coalescing properties. The only parameter in this case is the work-group size.

In the second codelet, multiple work-items in a work-group work together to load a strip of matrix data, and corresponding vector data, into shared memory. In this case, memory coalescing can be achieved. Work-group size and the size of the strip of data to be loaded are both parameters. Once the strip of data is loaded, a reduction-type computation needs to be performed. We assign one work-item per output element to perform the reduction.

The third codelet is similar to the second, except that the sum computation is performed in two stages by multiple work-items in a tree reduction.

4.3 Kernels for transpose

RaijinCL's GEMM implementation requires a high-performance implementation of transpose kernels. RaijinCL's GEMM implementation specifies the type of output memory object (buffer object or image datatype) and the vector length of the elements of output buffer, and asks our transpose API to return a kernel. Thus, we autotune a family of transpose kernels.

The simplest codelet considered by our autotuner is that each work-item picks element from input matrix and writes it to the transposed index in the output matrix. The only parameter in this case is the work-group size. However, this may not be optimal on some architectures. Transpose is a memory-bandwidth-bound kernel. For optimal memory bandwidth, some GPUs require that the accesses to memory be coalesced. Memory coalescing is achieved when multiple work-items in the same work-group access the same aligned block of memory, where the required block size is architecture-dependent. Thus, our second codelet implements a scheme where items in a work-group bring data into local memory in a coalesced manner. The work-items then perform the transposed load from the local memory instead of global memory.

5 Experimental results and analysis

To evaluate our auto-tuning library we measured the performance of the kernels across 5 GPU and 2 CPU devices, as described in detail in Section 5.1.

We first summarize the results, and then present a more detailed evaluation. The optimal parameters found for SGEMM, DGEMM and CGEMM are summarized in Table 1. We also report percentage of peak attained by RaijinCL and vendor BLAS (where available) on each architecture in the best case for each library in Table 2. Details of performance for the GPU devices, across different problem sizes are given in Figure 2 for SGEMM, Figure 3 for DGEMM, and Figure 4 for CGEMM.

The highlights are our results are:

- The GEMM routine generated by our library outperformed AMD's OpenCL BLAS on AMD GCN architecture, reaching over 3 TFlops on SGEMM and CGEMM.
- Our library was very close to the performance of CUBLAS on Nvidia GPUs.
- Our autotuner found different parameter settings for all architectures, but found that TN was the best suited layout for all GPUs.

Architecture Vendor Type	GCN AMD GPU	Evergreen AMD GPU	Fermi Nvidia GPU	Kepler Nvidia GPU	IB GPU Intel GPU	IB CPU Intel CPU	Shanghai AMD CPU
Layout	TN	TN	TN	TN	TN	NT	NT
Elements are consecutive	No	No	No	No	No	Yes	Yes
Tile size	(8,8,1)	(8,8,2)	(4,8,16)	(8,4,16)	(8,4,8)	(4,2,8)	(4,2,4)
Work-group size	(8,8)	(8,8)	(8,8)	(16,16)	(8,8)	(4,16)	(16,4)
SIMD width	4	4	4	2	4	8	4
Bring A to local memory	No	No	Yes	Yes	Yes	No	No
Bring B to local memory	No	No	Yes	Yes	Yes	No	No
Store A in image	No	No	No	No	No	No	No
Store B in image	No	No	Yes	No	No	No	No

(a) SGEMM

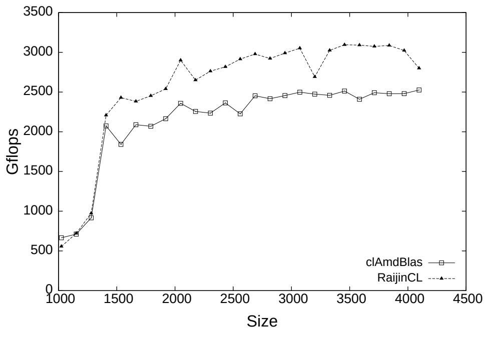
Architecture Vendor Type	GCN AMD GPU	Evergreen AMD GPU	Fermi Nvidia GPU	IB CPU Intel CPU	Shanghai AMD CPU
Layout	TN	TN	TN	NT	NT
Elements are consecutive	No	No	No	Yes	Yes
Tile size	(4,4,1)	(8,8,1)	(4,4,8)	(4,2,4)	(4,2,2)
Work-group size	(8,8)	(8,8)	(8,8)	(8,8)	(8,8)
SIMD width	2	2	2	4	2
Bring A to local memory	No	No	Yes	No	No
Bring B to local memory	No	No	Yes	No	No
Store A in image	No	No	Yes	No	No
Store B in image	No	No	Yes	No	No

(b) DGEMM

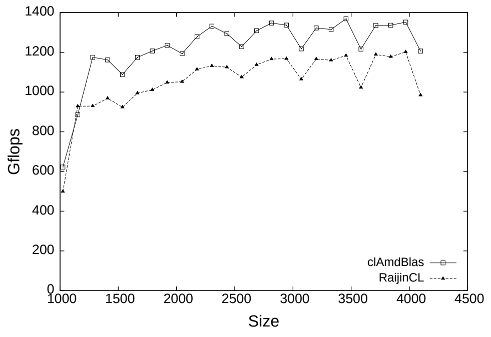
Architecture Vendor Type	GCN AMD GPU	Evergreen AMD GPU	Fermi Nvidia GPU	Kepler Nvidia GPU	IB GPU Intel GPU
Layout	TN	TN	TN	TN	TN
Elements are consecutive	No	No	No	No	No
Tile size	(4,4,2)	(4,4,2)	(4,4,8)	(4,4,8)	(4,4,4)
Work-group size	(8,8)	(8,8)	(8,8)	(16,4)	(4,16)
Complex vector size	2	2	2	2	1
Bring A to local memory	No	No	Yes	No	Yes
Bring B to local memory	No	No	Yes	No	Yes
Store A in image	No	No	Yes	Yes	No
Store B in image	No	No	Yes	Yes	No

(c) CGEMM

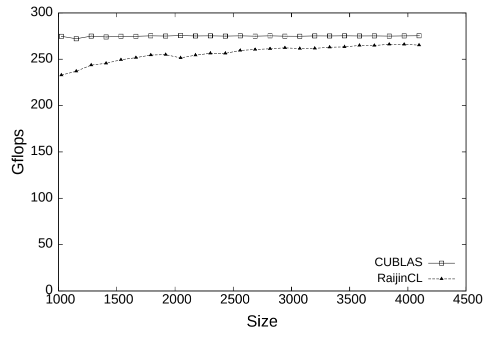
Table 1: Optimal Parameters for SGEMM, DGEMM and CGEMM



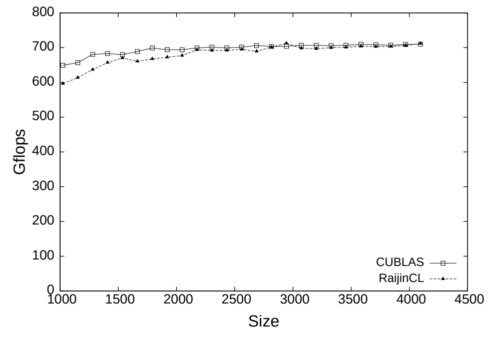
(a) AMD Graphics Core Next



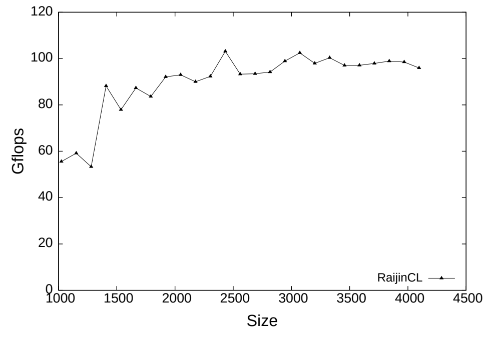
(b) AMD Evergreen



(c) Nvidia Kepler



(d) Nvidia Fermi

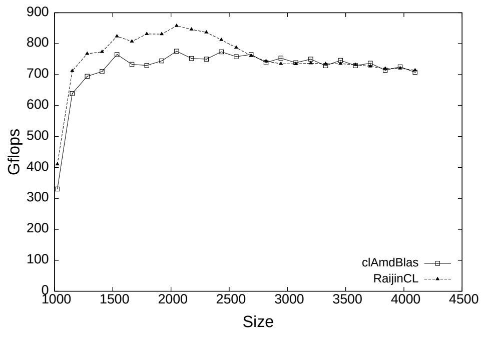


(e) Intel Ivy Bridge GPU

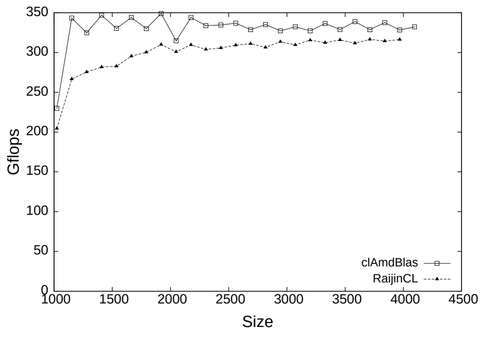
Figure 2: GPU performance for SGEMM

Architecture Vendor Type	GCN AMD GPU	Evergreen AMD GPU	Fermi Nvidia GPU	Kepler Nvidia GPU	IB GPU Intel GPU	IB CPU Intel CPU	Shanghai AMD CPU
SGEMM (RaijinCL)	81.6	57.5	69.0	36.1	36.6	51.9	52.4
SGEMM (vendor)	66.6	64.7	68.8	37.3	N/A	86.9	86.0
DGEMM (RaijinCL)	89.3	75.6	59.2	N/A	N/A	48.7	49.1
DGEMM (vendor)	81.6	82.0	61.1	N/A	N/A	87.5	90
CGEMM (RaijinCL)	79.2	64.3	77.3	41.4	41.2	N/A	N/A
CGEMM (vendor)	73.6	63.5	80.8	42.9	N/A	N/A	N/A

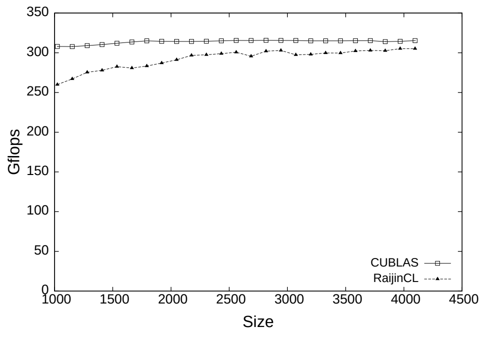
Table 2: Percentage of peak obtained on each device in best case



(a) AMD Graphics Core Next

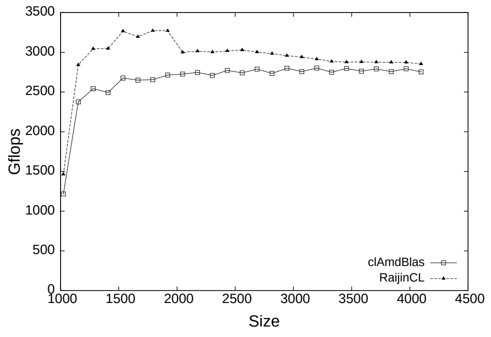


(b) AMD Evergreen

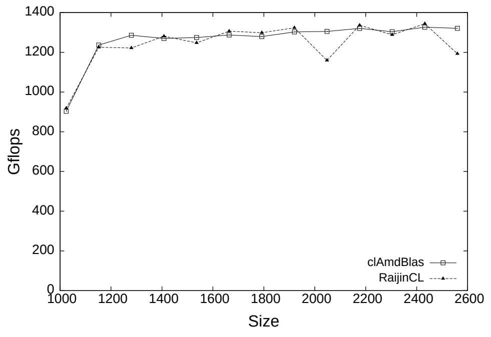


(c) Nvidia Fermi

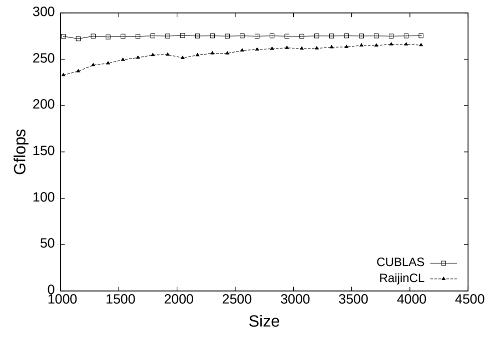
Figure 3: GPU performance for DGEMM



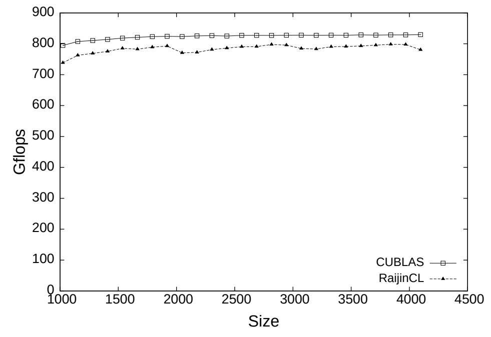
(a) AMD Graphics Core Next



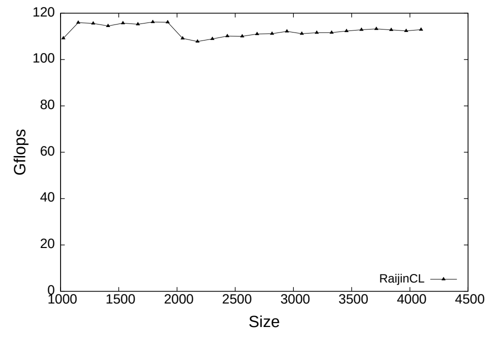
(b) AMD Evergreen



(c) Nvidia Kepler



(d) Nvidia Fermi



(e) Intel Ivy Bridge GPU

Figure 4: GPU performance for CGEMM

- The GEMM routine generated by our library was slower than an optimized BLAS on CPUs, but we still achieved about 50% of peak in SGEMM and DGEMM.

5.1 Architecture Tested

AMD Graphics Core Next architecture: We tested AMD Radeon HD 7970 GPU which is based upon the Graphics Core Next architecture. The machine configuration was Core i7 3820 CPU, 12GB DDR3 RAM, 2x Radeon HD 7970 2GB, Kubuntu 12.04 and Catalyst 13.4 GPU drivers. Performance was compared against AMD's OpenCL BLAS APPML v1.10. AMD's BLAS comes with its own auto-tuner, and this tuner was run before the experiments.

AMD Evergreen architecture: We tested AMD Radeon HD 5850 GPU based upon AMD Evergreen architecture. The machine configuration was AMD Phenom II x4 925 CPU, 8GB DDR3 RAM, Radeon HD 5850 1GB, Kubuntu 12.04 and Catalyst 12.10 GPU drivers. Performance was tested against AMD's OpenCL BLAS v1.10. AMD's auto-tuner was run before the experiments.

Nvidia Fermi architecture: We tested Tesla C2050 GPU based upon Nvidia Fermi architecture. The machine configuration was Core i7 920 CPU, 6GB DDR3 RAM, Tesla C2050 3GB, GTX 480 GPU, Ubuntu 12.04 and Nvidia 295.13 drivers. Performance Performance was compared against CUBLAS v4.1.

Nvidia Kepler architecture: We tested Nvidia GT650M GPU based upon the Nvidia Kepler architecture in a laptop with a Core i7 3610QM CPU, 6GB DDR3 RAM, Windows 7 x64 and Nvidia 314.07 drivers. Our GT650M unit has a clockspeed of 835MHz with turbo, and equipped with 2GB GDDR5 RAM with 64GB/s of bandwidth. Performance was compared against CUBLAS v5.0.

Intel Ivy Bridge GPU architecture: The machine used for this test was the same as the machine used for Nvidia Kepler test, except that the Intel Ivy Bridge GPU [6] HD 4000 was tested. We used Intel driver build version 9.18.10.3071. Intel does not supply a BLAS for the Ivy Bridge GPU.

Intel Ivy Bridge CPU architecture: We tested Core i7 3610QM CPU based upon Ivy Bridge architecture. The hardware configuration is the same as mentioned in Ivy Bridge GPU section. Software platform was Kubuntu 12.04 64-bit and we used OpenCL CPU implementation shipped with AMD's APP SDK 2.8. Performance was compared against OpenBLAS [15]

AMD Shanghai CPU architecture: The configuration mentioned is the same as that mentioned in the AMD Evergreen GPU tests. We used the OpenCL CPU implementation shipped with AMD's APP SDK 2.8. Performance was compared against OpenBLAS.

5.2 Discussion of Results

We first discuss some architecture-independant issues and then discuss results on each architecture separately.

During the development of the library, we investigated FMA (fused multiply-add) instructions. OpenCL supports FMA as a built-in function. Some architectures provide a FMA instruction in hardware. OpenCL provides a preprocessor macro FP_FAST_FMAF for OpenCL kernels. If the implementation supports a fast FMA operation, then the macro should be defined. However, we discovered that some implementations do define the macro, but our kernels achieved 10 to 20% lower performance when using the FMA built-in on

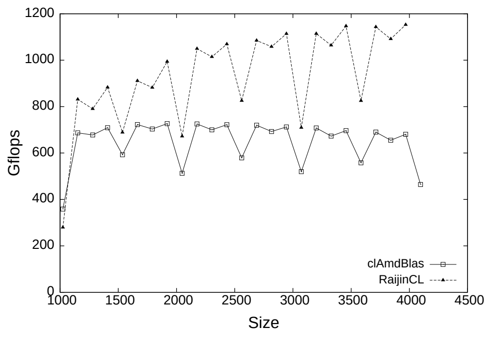


Figure 5: SGEMM performance on Evergreen on NT kernel, internally converted to TN by RaijinCL

single-precision operations. Now we do not generate calls to FMA for single-precision on any architecture, and only generate them for double-precision if the implementation defines FP_FAST_FMA.

Now we look at individual architectures, starting with AMD's Graphics Core Next (GCN) architecture. We compared performance of RaijinCL to AMD's OpenCL BLAS on SGEMM, DGEMM and CGEMM, shown in Figure 2 (a), Figure 3(a) and 4(a) respectively. RaijinCL significantly outperforms AMD's OpenCL BLAS on all GEMM kernels and reached a peak of about 3.1 teraflops on SGEMM. As shown in Table 1, we found that the best performing kernels on GCN do not utilize local memory and do not copy data into images.

On AMD's prior-generation Evergreen architecture, RaijinCL again found that the best kernels do not utilize local memory. Previously, some researchers, such as Du et al. [5], had noted that copying data into images is required for a high-performance GEMM implementation on Evergreen. However, the best kernels found by RaijinCL for Evergreen shown in column 2 of Table 1 a,b and c do not utilize images. The discrepancy can be explained. Evergreen has a L1 data cache for read-only data which is quite important for performance on GEMM. Previously, this L1 cache was only used for image data but recent improvements in AMD's drivers now enable use of this cache for OpenCL buffers as well.

AMD's OpenCL BLAS was slightly faster than RaijinCL on SGEMM and DGEMM on TN kernels. However, we also found that AMD's library shows a performance drop on inputs not in the TN layout. As discussed, RaijinCL's autotuner found that TN was the best layout for Evergreen. After tuning, when the user calls RaijinCL's GEMM API on Evergreen with parameters other than TN, RaijinCL transforms the input to the optimal format internally. We tested the performance of RaijinCL (post-tuning) and AMD's BLAS on SGEMM NT layout and performance is shown in Figure 5. RaijinCL outperformed AMD's library significantly in this case.

On Nvidia Fermi architecture, RaijinCL's performance is very close to the vendor's CUBLAS on SGEMM and DGEMM(Figures 2(c),3(c) and 4(c)). We found that using local memory for both operands was highly desirable. On SGEMM, while the best kernel found by RaijinCL utilized OpenCL images, the performance boost of using OpenCL images over using buffers was minimal in this case. However, for DGEMM, using OpenCL images gave a performance boost of almost 40% compared to not using images.

On Nvidia Kepler architecture, we again found that the highest performing SGEMM kernels utilized local memory. Both RaijinCL and CUBLAS only achieved about 40% of theoretical peak. We suspect that the performance of SGEMM is limited by memory bandwidth from local memory. Kepler GPU is divided into cores called SMX units. Each SMX has 192 ALUs each capable of one FMA instruction per cycle, but the associated local memory can only service 32 floats per cycle. The tile size of the best found kernel was

8×4. This kernel will perform 12 loads from local memory for every 32 multiply-adds. However, given the 32 floats/cycle limit on bandwidth, the instruction throughput is theoretically limited to only 44%, thus showing the memory bandwidth limitation. On CGEMM, we found that the best kernel did not utilize local memory at all and instead relied upon loading data directly into registers from images. This is in contrast to the general recommendation that GEMM-type kernels should use local memory on Nvidia GPUs.

Intel Ivy Bridge GPU (HD 4000) architecture is not well known in high performance computing related literature, and thus we provide a brief description. HD 4000 has 16 execution units, and each EU has two 4-wide SIMD pipes, and each pipe can perform 4 single-precision multiply-accumulate (MAC) operations. Thus, each EU has a theoretical peak of 16 flops/cycle. The HD 4000 SKU in our machine is clocked at 1.1GHz, and has a peak of 281.6 GFlops. The 16 EUs share 128kB of local memory, which can provide a sustained read bandwidth of upto 128 bytes/cycle.

Intel does not provide a BLAS for the HD 4000 that we could compare against, but some insight into RaijinCL's performance can be gained from the percentage of peak attained. RaijinCL achieved about 36.5% on SGEMM (Figure 2(e)), and more than 40% on CGEMM(Figure 4(e)). We found that using image datatype typically gave very bad performance compared to buffers. Intel‘s method of mapping of work-items to EUs is another interesting issue. If the kernel is performing operations with SIMD width less than the SIMD width of the EU, then multiple work-items may get mapped to individual ALU pipes in the EU. However, if the kernel code is predominantly using vector width matching the EU width, then a single work-item may occupy the full width of the EU. Our hypothesis is that the former case happens for the best-found CGEMM kernel, and the latter case happens for SGEMM.

We also compared the performance of RaijinCL with OpenBLAS on AMD and Intel CPUs. We found that RaijinCL was much slower than OpenBLAS but delivered around 40% of peak. We examined the assembly generated by OpenCL CPU drivers and found that the OpenCL compiler successfully mapped the SIMD constructs in OpenCL to SIMD instruction set of CPUs for our kernel. However, we also noticed that AVX and SSE registers were poorly utilized, with compilers generating some unnecessary move instructions as well as unnecessary register spills. Thus, there remains significant room of improvement in OpenCL compilers for CPUs. We also experimented with an alternate strategy for our code-generator. Instead of generating work-groups of size, say $(8, 8)$, we collapsed the work-group into a single item by inserting an explicit outer loop in the kernel body. The idea was that a work-group is typically mapped to a single CPU core, and thus there is no need for more fine-grained parallelism. However, we found that the performance of collapsed and non-collapsed work-groups was nearly identical.

6 Related Work

Autotuning is a well-established technique on CPUs. ATLAS [3] is a well-known example of an autotuning BLAS. Autotuning has also been used for some FFT libraries such as FFTW. However, FFT libraries typically used an online tuner where the user application first creates a plan and then executes the plan. The autotuner is called by the plan creation routine whereas libraries like ATLAS perform autotuning at install time and thus the applications using the application do not need to call any plan creation routines. Our approach is similar to ATLAS in this regard.

Implementing a fast GEMM routine on accelerators has been of considerable interest in the past few years. Several researchers have written hand-tuned implementations for particular GPU architectures Volkov et al. [13] described a fast GEMM prototype in CUDA for Nvidia's Tesla architecture. Their ideas have now been included in Nvidia's CUBLAS library. Nakasato [9] described a fast GEMM prototype for AMD's

Cypress GPU (which powered products such as Radeon 5870). Their implementation was written in AMD's CAL API, which was a low-level pseudo-assembler exposed by AMD for their previous-gen GPUs. CAL API has now been deprecated. Nath et al. [10] report a fast CUDA GEMM implementation for Nvidia Fermi GPUs. Tan et al. [12] describe a fast CUDA GEMM implementation for Nvidia's Fermi architecture. They wrote their implementation in PTX, which is a pseudo-assembler for CUDA. This allowed tighter control over instruction scheduling compared to high-level languages like CUDA-C and OpenCL. They report better performance than CUBLAS. Matsumoto et al. [8] reported several high-performance OpenCL GEMM kernels for AMD's Tahiti GPU. Their implementation uses an autotuner to search for parameters, though they limit their experimental evaluation to only one architecture so it is not clear how well it will translate to other architectures. Schneider et al. [11] implemented a fast ZGEMM routine on Cell Broadband Engine that ran on Cellś Synergistic Processing Unit (SPU). They optimized their implementation for the vector instruction set of the Cell processor.

Several researchers have looked at portable OpenCL GEMM implementations for multiple architectures. Du et al. [5] present a study of a portable GEMM impelementation. They had two codepaths. One codepath was an OpenCL translation of Fermi GEMM routines from MAGMA's CUDA kernels. This was essentially a handwritten implementation with only a few parameters. Second codepath was an autotuning implementation for AMD GPUs. In comparision, our library offers a unified and completely autotuning implementation for all architectures. In terms of performance, our implementation offers about 10% higher performance on SGEMM on Nvidia GPUs, and similar performance on AMD GPUs. They did not test their library on CPUs. Weber et al. [14] discussed an autotuning implementation on AMD's GCN GPUs. However, their reported results for AMD Radeon 7970 (1.7 teraflops on SGEMM and 650 teraflops on DGEMM) are much slower than our results.

7 Conclusions

OpenCL is emerging as a common, portable programming language for a wide variety of compute devices available from many different vendors. However, even though OpenCL is portable, different devices require different OpenCL implementations of common BLAS routines in order to achieve high performance. In this paper we presented a solution to this problem via the design, implementation and evaluation of RaijinCL, a portable and practical autotuning OpenCL library for GEMM and other matrix operations.

In designing RaijinCL we aimed for a solution that was both autotuning and easy to deploy. Users of the library only need to tune the library once, by either running our command utility (which only requires the OpenCL driver), or by simply reusing a previously generated device profile. We have generated profiles for the devices we have experimented with, and we hope the community and hardware vendors will contribute more.

Our solution follows the asynchronous design of OpenCL, so that RaijinCL provides an asynchronous API whereby calls to the OpenCL device will not block computations on the CPU.

A further design decision was to provide both a high-level API, which is similar to other GPU BLAS APIs, as well as a lower-level API which exposes temporary buffers to the API user. This allows the user of the API to know more precise memory requirements and to handle the reuse of memory between several API calls.

Our autotuning approach was designed by identifying a collection of codelets for each kernel. Different versions of the codelets expose important algorithmic variations. For example, for GEMM the codelet variants expose the argument layout (transposed or not) and the element assignment scheme (consecutive or offset). Within each codelet we identified important parameters such as tile sizes, SIMD width, work-group size, how to handle local memory and whether or not to use OpenCL images. Given the group of codelets and the parameter space, the autotuner evaluates all points in the search space and identifies the best codelet, and the best parameters for that codelet.

We experimented on a wide variety of devices, including 5 GPUs and 2 CPUs, including devices from AMD, Nvidia and Intel. We found that the autotuner did find different parameter settings for different devices, so our choice of search parameters seems to be reasonable.

In terms of performance, for each device we compared our autotuned library to the vendor's specialized library (when one was available). We found that for the GPUs we sometimes outperformed the vendor's library and we never under-performed by a significant margin. Thus, we did achieve the goal of having a portable and high-performance solution across a range of GPUs. The performance on the CPUs was reasonable, but vendor's specialized CPU libraries were significantly better. We hope that as the OpenCL compilers get better, some of that gap will be reduced.

RaijinCL is open source, and we hope that users will use it on many different devices. As we gain more experience and feedback from users we may be able to further tune to library by exposing further codelets and additional parameters.

References

[1] The OpenCL Specification. `http://www.khronos.org/opencl`.

[2] AMD. AMD Accelerated Parallel Processing Math Libraries. `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/`.

[3] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[4] NVIDIA Corp. Nvidia CUBLAS library. `http://developer.nvidia.com/cublas`.

[5] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012.

[6] David Kanter. Intel Ivy Bridge Graphics Architecture. `http://www.realworldtech.com/ivy-bridge-gpu/`.

[7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[8] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukin. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. Technical Report 2012-002, Graduate School of Computer Science and Enginering, The University of Aizu, July 2012.

[9] Naohito Nakasato. A fast GEMM implementation on the Cypress GPU. *SIGMETRICS Perform. Eval. Rev.*, 38(4):50–55, March 2011.

[10] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved Magma GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, November 2010.

[11] T. Schneider, T. Hoefler, S. Wunderlich, T. Mehlan, and W. Rehm. An optimized ZGEMM implementation for the Cell BE. In *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA)*, Dresden, Germany, February 2008.

[12] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[13] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] Rick Weber and Gregory Peterson. A trip to Tahiti: Approaching a 5 Tflop SGEMM using 3 AMD GPUs. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012*, 2012.

[15] Zhang Xianyi, Qian Wang, and Zhang Yunquan. Model-driven level 3 BLAS Performance Optimization on Loongson 3A Processor. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, December 2012.